RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

D5.1 (WP5): Offline SD Erlang Profiling Tool

Due date of deliverable: 30th September 2012
Actual submission date: 30th September 2012

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: University of Kent Revision: 0.1

Purpose: To design and deliver an offline profiling tool reporting monitoring and visualization results in static reports and interactively.

Results: The main results of this deliverable are

- Percept2, a version of the Percept tool enhanced with functionality to support multicore and distributed systems, and also refactored to support scalability.
- Extension of the built-in Erlang tracing mechanism to support scalability.
- Sampling-based profiling of Erlang systems.
- Extension of the DTrace/SystemTap system for Erlang including new probes, and reports.

Conclusion: We have delivered a first set of tools to support profiling for SD Erlang.

Project funded under the European Community Framework 7 Programme (2011-14)			
Dissemination Level			
PU	Public		⁎
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Offline SD Erlang Profiling Tool

Contents

1 Executive Summary

This report presents the first version of the deliverable D5.1 for the RELEASE project. The aim of this deliverable is to provide tools to support the offline understanding and tuning of massively multicore systems written in Scalable Distributed Erlang (SD Erlang).

This first release has concentrated on tools supporting the multicore implementation of Erlang, while also providing support for aspects of distribution. A second release, in month 24 of the project, will provide explicit support for the features of SD Erlang. The main results of this deliverable are

- Percept2, a version of the Percept tool enhanced with functionality to support multicore and distributed systems, and also refactored to support scalability.
- Extension of the built-in Erlang tracing mechanism to support scalability.
- Sampling-based profiling of Erlang systems.
- Extension of the DTrace/SystemTap system for Erlang including new probes, and reports.

2 Introduction

At the heart of the RELEASE project is the development of SD Erlang, that uses Erlang distribution to handle large-scale multicore systems in Erlang. SD Erlang has been designed, and will be implemented in the second year of the project.

To build SD Erlang systems of any size, it is necessary to monitor and visualise the behaviour of these systems, and Work Package 5 of RELEASE is designed to do this. While many-core systems in SD Erlang will necessarily be distributed, it is also the case that they will contain single multicore processing elements that deploy Erlang. Monitoring and visualising SD Erlang will therefore necessarily require in-node observation as well as monitoring what goes on between nodes.

The deliverable D5.1 outlines the progress of the project in meeting this goals, and in particular describes in detail the innovations that have been made in *offline monitoring and visualisation* for Erlang multicore and distributed systems. These include enhancements of tools within the Erlang ecosystem, including Percept, the built-in facilities for Erlang tracing, and further development of the facilities for tracing by means of DTrace/SystemTap.

These constitute the first delivery of offline monitoring tools, and a second delivery will be made in M24, complementing the delivery of the implementation of SD Erlang itself in M18.

The rest of this report is structured as follows. Section 3 covers background and related work in tracing and profiling for Erlang, and describes both the foundational technologies (Erlang built-in tracing, DTrace) and tools already built using these. Section 4 describes in detail how the Percept tool has been extended by the project to *Percept2*. Section 5 describes the extensions provided by the project to DTrace/SystemTap, and how these are used in profiling and visualisation, and Section 6 describes a number of other contributions, before future work is summarised in Section 7.

3 Related work

3.1 Erlang built-in tracing

The Erlang [CT09, Arm10] runtime system has built-in support for tracing many types of events. With the built-in tracing, an Erlang program can be traced while being executed, and no special compilation or instrumentation of the program is needed.

Erlang's built-in support for tracing is exposed to users through a number of built-in functions (BIFs), i.e. `erlang:trace/3`, `erlang:trace_pattern/3`, `erlang:trace_info/2`, etc. The

function `erlang:trace/3` *enables* and *disables* the low-level tracing. When enabling the tracing, the user can specify which processes to trace, and which events they are interested in. The process that makes the call to `erlang:trace/3` to enable the tracing is known as the *tracer process*. In Erlang, at any one time, a process can only be traced by one tracer process.

Events that can be traced include: global and local function calls, process-related activities, message passing, garbage collection and memory usage.

When tracing is enabled, trace events are sent as messages of the following format:

```
{trace, Pid, Tag, Data1 [,Data2]}
```

where `[,Data2]` denotes an optional field dependent on the trace message type. If the timestamp flag is given, the first element of the tuple will be `trace_ts` instead and the timestamp is added last in the tuple.

The `erlang:trace_pattern/3` BIF, used in conjunction with `erlang:trace/3`, is for enabling the tracing of local and global function calls. With `erlang:trace_pattern/3`, a user can specify the subset of functions to be traced using Erlang's `match specification`. A function `call/return_to` trace event will be generated only if a *traced* process executes a *traced* function.

Erlang's built-in tracing is powerful, but not without limitations:

- first, because a traced process can only have one tracer process at any one time, it is impossible to have several tools requiring trace information run concurrently on a node;
- second, while *match specification* can be used to have a fine control over what kind of function `call/return_to` trace events to generate, the similar kind of dynamic filtering support is not available to other trace events. Aggregation of trace results is not supported either;
- third, there is no support for remote or distributed tracing, that is all settings have to be executed on the node and the trace process has to be node local;
- finally, tracing too much adds overhead, and could slow down the application being traced significantly.

Sequential Tracing. Apart from the tracing support provided by `erlang:trace/3`, Erlang provides another facility especially for tracing all messages resulting from one initial message, which is known as *sequential tracing*.

Sequential tracing is a way to trace a sequence of messages sent between different local or remote processes, where the sequence is initiated by one single process. In Erlang, each process has a *trace token*, which can be empty or not empty. The trace token is passed invisibly with each message. To start a sequential trace, the user explicitly sets the trace token in the process that will send the first message in a sequence. The trace token of a process is set each time the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not.

On each Erlang node a process can be set as the system tracer. This process will receive trace messages each time a message with a trace token is sent or received (if the trace token flag `send` or `receive` is set). The system tracer can then print each trace event, write it to a file or whatever suitable.

3.2 Tracing and Profiling Tools

The Erlang built-in functions for tracing are powerful, but they are also very low-level and are not very user-friendly. In practice, a sequence of function calls are needed to setup an interesting trace. Apart

from that, there are no ready-to use BIFs from Erlang for the display and analysis of trace results. Unsurprisingly, a collection of tracing/profiling tools have been built on top of Erlang's built-in trace, and most of these tools are part of the standard Erlang distribution. A brief description of each of these tools follows.

3.2.1 Erlang tracing tools

The main Erlang tracing tools are discussed here.

- **Pman - the Process Manager**. Pman [pma] is a graphical tool built on top of the Erlang built-in trace for inspecting the state of processes in Erlang systems. With Pman it is also possible to trace events in the individual processes, such as the current function the process is executing, the number of messages in the process's mailbox, etc.

- **The DBG tracer**. Complement to Pman, the *dbg* [dbg] tracer is a text-based debugger providing a user-friendly interface to the `trace` and `trace_pattern` BIFs. To eliminate the complexity of writing complex match specifications for tracing functions, DBG provides a function called `dbg:fun2ms/1`, which converts specifications that are described using `fun` syntax into match specifications.

- **ET - the Event Tracer**. ET [et] also uses the built-in trace mechanism in Erlang, but designed for displaying message sequence charts for Erlang applications. What distinguishes ET from other tracing tools is that ET requires the user to instrument their code with calls but it is designed for displaying message sequence charts for Erlang applications. What distinguishes ET from other tracing tools is that ET requires that users instrument their code with calls to the function `et:trace_me/5` in strategic places where there is interesting information available in the program. A call to `et:trace_me/5` may be something like:

  ```
  et:trace_me(85, from, to, message, extra_stuff)
  ```

  Those calls to `et:trace_me/5` in the user code are then traced, and information is collected from the arguments of these function calls. The information extracted is then used to derive the message sequence charts.

- **TTB - The Trace Tool Builder**. TTB [ttb] is a base for building trace tools for single node or distributed erlang systems. The main features of the TTB are:

  - Start tracing to file ports on several nodes with one function call.
  - Write additional information to a trace information file, which is read during formatting.
  - Some simple support for sequential tracing.
  - Restoring of previous configuration by maintaining a history buffer and handling configuration files.
  - Formatting of binary trace logs and merging of logs from multiple nodes.

- **ETop - The Erlang Top**. ETop [eto], like Pman, is also a tool for presenting information about Erlang processes, but with an aim to present information about Erlang processes similar to the information presented by *top* in UNIX. The output from ETop can be graphical or text based.

3.2.2 Erlang Profiling tools.

Erlang/OTP contains server profiling tools for finding performance bottlenecks in Erlang programs, as described next.

- **fprof**. `fprof` [fpr] measures the execution time for each function, both own time, i.e. how much time a function has used for its own execution, and accumulated time, i.e. including called functions. `fprof` also gives how many times each function has been called. The values are displayed per process. Compared to other function profiling tools, `fprof` provides the most detailed information about where time is spent, but it significantly slows down the program it profiles.

- **eprof**. Different from `fprof`, which is based on `erlang:trace/3` and `erlang:trace_pattern/3`, `eprof` [epr] is based on the `erlang:trace_info/3` BIF, which returns trace information about a process or function. `eprof` shows how much time has been used by each process, and in which function calls this time has been spent. `eprof` does not generate function callgraphs and has considerable less impact on the program profiled.

- **cprof**. `cprof` [cpr] only counts how many times each function is called when the program is run, on a per module basis. It has a small slowdown effect on the application being profiled.

- **lcnt - The Lock Profiler**. `lcnt` [lcn] is not based on the Erlang built-in tracing, instead it is a tool used to profile the internal ethread locks in the Erlang Runtime System. With lcnt enabled, internal counters in the runtime system are updated each time a lock is taken. The counters stores information about the number of acquisition tries and the number of collisions that has occurred during the acquisition tries. The counters also record the waiting time a lock has caused for a blocked thread when a collision has occurred.

- **Percept - the *er*lang *c*oncurr*e*ncy *p*rofiling *t*ool**. Percept [per] is a tool to visualise Erlang application level concurrency and identity concurrency bottlenecks. It utilises `erlang:trace/3` and `erlang:system_profile/2` to monitor events from process states, i.e. waiting, running, runnable, free and exiting. A waiting or suspended process is considered an inactive process and a running or runnable process is considered an active process.

  In Percept, events are collected and stored to a file. The file can then be analysed; the analyser parses the data file and inserts all events in a RAM database, `percept_db`. Once the analysis is done, the data can be viewed through a web-based interface.

  Percept generates an application-level concurrency graph, as the example shows in Fig 1. This graph shows the number of active processes at any one time during the profiling; dips in the graph represents low concurrency. One can zoom in on different areas of the graph to get a clear picture for a specific time interval.

  Clicking on the *processes* button from the menu leads to a page showing the process table, as shown in Fig 2. Each table row shows information about a process, including a lifetime bar that presents a rough estimate in green color about when the process was alive during profiling, an entry-point, its registered name if it had one and the process's parent id. Process ids in this table are click-able, and clicking on a process id direct you to the process information page for this process, as shown in Fig 3. In the process information pages, Percept also shows a process's inactive times: how many times it has been waiting and in which function.

  In the processes page, it is possible to select a number of processes of interest by ticking the *select* box, as shown in Fig 2, and compare their runnability during the execution by pressing the *compare* button.

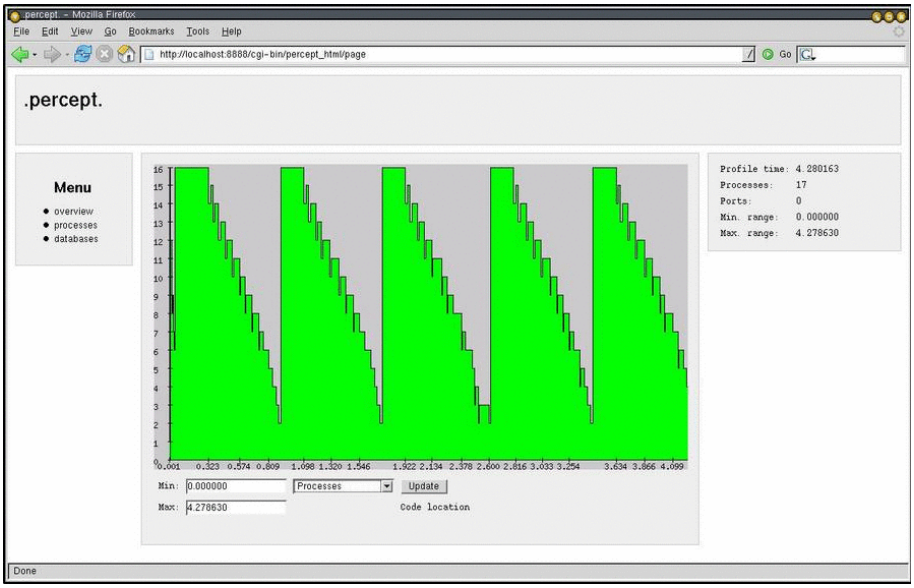

Figure 1: Percept: concurrency overview

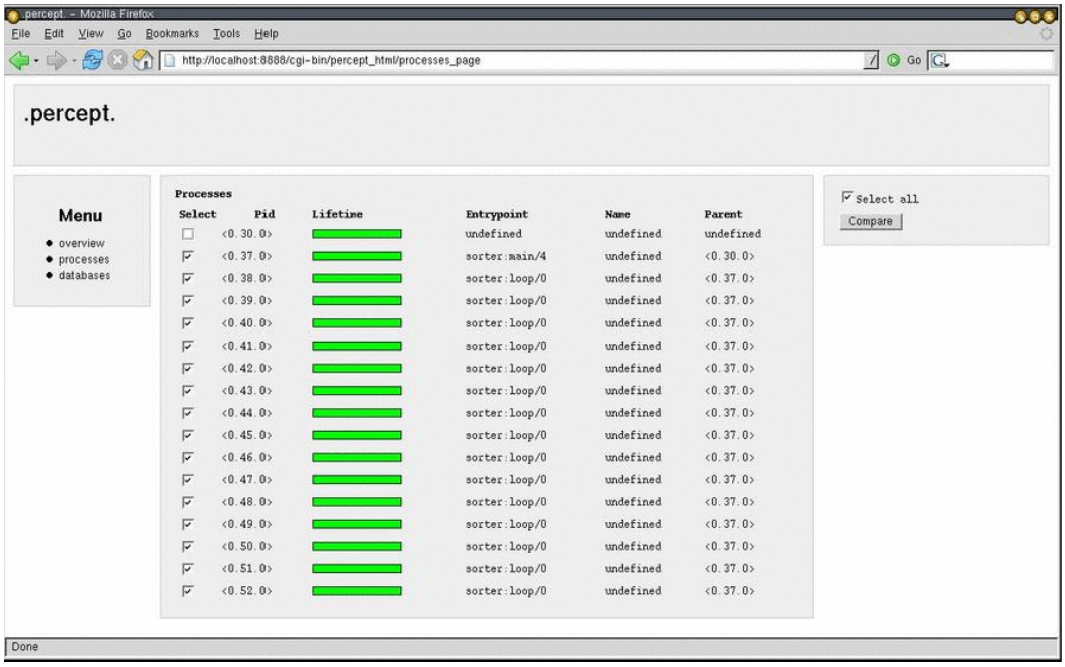

Figure 2: Percept: process selection

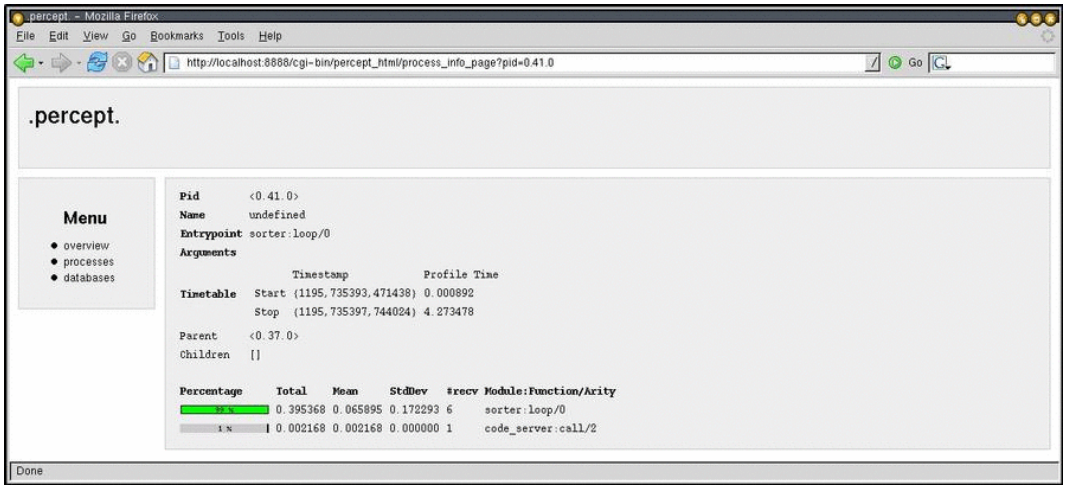

Figure 3: Percept: process information page

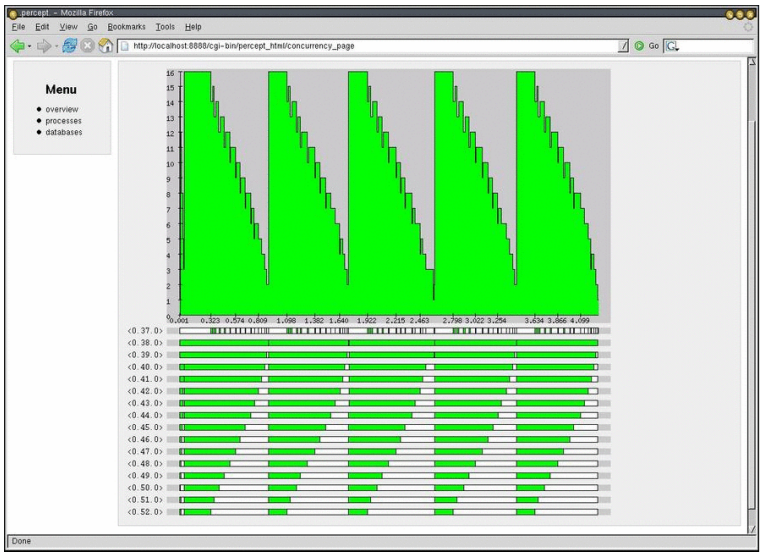

Figure 4: Percept: process comparison

As shown in the *compare* page in Fig 4, the activity bar under the concurrency graph shows each process's runnability. The green color shows when a process is active (which is running or runnable) and the white color represents time when a process is inactive (waiting in a receive or suspended).

3.3 DTrace and SystemTap

The increasing complexity of computing systems and especially the introduction of multicore technology to a wide range of users have led to the need to identify performance issues of a running system at all levels of its software stack. Ideally, this analysis should be performed on a live, running system and not on information that is first gathered and then analyzed. Furthermore, only information regarding the observed problems should be collected, in order to minimize overhead and interaction with the measurement infrastructure. **DTrace** is a dynamic tracing framework that provides such a functionality. It can be used by administrators and developers alike to examine the behavior of applications and the operating system during development or even on live production systems. DTrace provides the functionality to understand how a system works, track down performance problems across many layers of software and locate the cause of spurious errors.

In comparison to other similar tools and instrumentation frameworks, DTrace presents several advantages. It is relatively lightweight, not imposing such a severe overhead to the running system. It does not require special recompiled versions of the software to be examined. Also, it does not require different tools to provide a complete view of system behaviour, nor significant post-processing to create meaningful information from the gathered data.

DTrace was initially developed for the Solaris operating system; it was however soon ported to OpenSolaris, NetBSD, FreeBSD and Mac OS X. A Linux port is also available but does not provide the full functionality of DTrace. **SystemTap** is a tool that is similar to DTrace, developed specifically for Linux. It provides a compatibility layer with DTrace; this allows to annotate the source code once and, depending on which system it will run, use either DTrace or SystemTap to collect the required data. SystemTap has been used in the context of RELEASE due to the fact that the Linux port of DTrace does not support all features of other supported architectures.

3.3.1 Features and architecture of DTrace and SystemTap

DTrace and SystemTap support both static and dynamic instrumentation. Software developers can introduce instrumentation points in the code. When disabled, these points contain no-operation (NOP) instructions that have no effect when executed and introduce a negligible runtime overhead. When enabled, instrumentation points correspond to **probes** that will **fire** when execution reaches them. Probes can be enabled dynamically in the software, as it is being executed.

When a probe fires, it is up to the person analyzing the system to determine what happens. In general, an appropriate **script** is executed, written in a special high-level control language that is called **D**. This script can collect and aggregate data, manage time stamps, collect stack traces, etc. It is up to the person analyzing the system to provide a meaningful set of D-scripts that will be executed, when probes fire in the system under inspection. A **predicate** mechanism can be used to specify that D-scripts need to be called only when certain conditions are satisfied. In this way, the person analyzing the system has full control over the information that is collected.

The overall architecture of DTrace is shown in Figure 5, which presents the main components of the DTrace infrastructure and their interactions [GM11]. **Consumers** are the applications that utilize the DTrace framework by calling into the routines in the DTrace library. Programs such as `dtrace` (or `stap` for SystemTap) are general-purpose consumers, whose behaviour can be specified using D-scripts. **Providers** are libraries of probes that provide logical abstractions of complex areas of the system. In

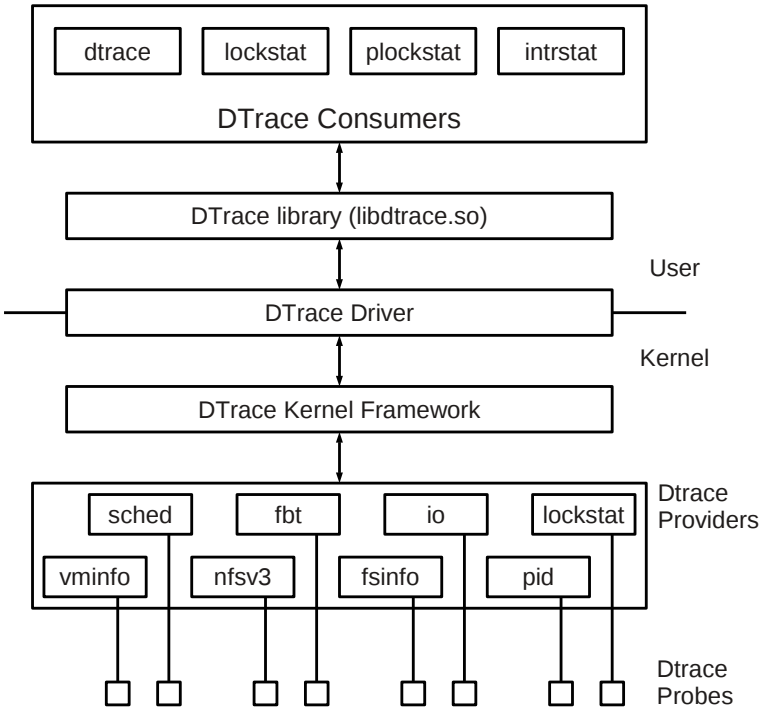

Figure 5: Overview of the DTrace Architecture

essence, they define instrumentation points that can be later enabled and be associated with D-scripts. In the context of the RELEASE project, the Erlang VM is the most interesting provider.

3.3.2 DTrace/SystemTap and Erlang/OTP

In the context of RELEASE, the importance of DTrace and SystemTap is twofold. First, they can be used to identify bottlenecks in the Erlang VM itself. The performance of the VM is of extreme importance, since every construct and operation of an Erlang program has to go through the VM, which performs the required low-level work. Hence, it became early obvious that analyzing the performance of the VM would be essential. This use of Dtrace and SystemTap is important for WP 2 of RELEASE and will not be discussed further here. Second, DTrace and SystemTap can be used to analyze and profile user applications running in the Erlang VM. This is especially important for WP 5 of RELEASE and this is where we will focus our attention in this report.

The first effort to annotate the Erlang VM source code (which is written in C) using DTrace probes was in 2008 by Garry Bulmer and Tim Becker [Bul08]. Their work was performed on the Erlang/OTP R12, but was later discontinued. Another effort was initiated in 2011 by Scott Lystig Fritchie [Fri11]. His work was finally merged into the official Erlang/OTP source code for release R15B01. According to the Erlang/OTP documentation, one of the goals of the current implementation is to annotate as much of the Erlang VM as is practical. Special attention must be paid to file I/O operations performed in the Erlang VM. The fact that an I/O worker pool is used makes it much more difficult to trace I/O activity, since a number of different operations can be requested by different threads of that pool. In essence, the whole I/O driver of the VM has been annotated with DTrace probes. The same holds for the callback API of drivers. Another specific goal of the current implementation is to allow additional annotations in Erlang code.

Other parts of the VM have not yet been annotated to the degree that would be desirable to identify errors or performance bottlenecks. Processes, for example, are currently probed only for very basic operations, like when they are spawned or when they exit. The scheduler that handles the processes

is actually not probed for the operations it performs, raising an opportunity to further investigate this front. Especially when the SMP scheduler is enabled, a number of interesting probes can be inserted into the code: measure run-queue length per processor, measure number of processes moved during work stealing, measure attempts to gain a lock of a run-queue and how many succeeded immediately, etc. A similar situation exists in other modules of the VM, like the garbage collector, the exchange of messages and data copy. Again of great interest to RELEASE are ETS tables and their performance when SMP is enabled in the VM. These are only a few examples of the many places where probes could provide clues about the performance of the Erlang VM.

4 Extending Percept: Percept2

4.1 Introduction

In this section we introduce *Percept2*, an enhanced version of *Percept*. Percept2 extends Percept in two aspects: functionality and scalability. Among the new functionalities added to Percept are:

- Scheduler activity: the number of active schedulers at any time during the profiling.
- Process migration information: the migration history of a process between run queues.
- Statistics data about message passing between processes: the number of messages, and the average message size, sent/received by a process.
- Accumulated runtime per-process: the accumulated time when a process is in a running state.
- Process tree: the hierarchy structure indicating the parent-child relationships between processes.
- Dynamic function call graph/count/time: the hierarchy structure showing the calling relationships between functions during the program run, and the amount of time spent in a function.
- Active functions: the functions that are active during a specific time interval.
- Inter-node message passing: the sending of messages from one node to another.

The following techniques have been used to improved the scalability of Percept.

- Compressed process tree/function call graph representation: an approach to reducing the number of processes/function call paths presented without losing important information.
- Parallelisation of Percept: the processing of profile data has been parallelised so that multiple data files can be processed at the same time.
- Caching of history web-pages.

4.2 Percept2 by example

In the rest of this section, we describe how to use Percept2, and the functionalities Percept2 supports.

4.2.1 Profiling

There are a few ways to start the profiling of a specific code. The function `percept2:profile/3` is the preferred way.

`percept2:profile/3` takes 3 parameters. A file specification for the data destination as the first argument. The file specification can be a filename (which is the case for Percept) or a wrap file

specification as what is used by the dbg library. If a single filename is specified, all the trace messages are saved in this file; if a wrap file specification is used, then the trace is written to a limited number of files each with a limited size. The actual filenames are `Filename ++ SeqCnt ++ Suffix`, where `SeqCnt` counts as a decimal string from `0` to `WrapCnt`.

With the current version of Percept2, if the number of files in this wrap trace is as many as `WrapCnt` the oldest file is deleted and a new file is opened to become the current (as what is described in dbg). For off-line profiling, this means some profiling data may get lost. We are in the process of addressing this problem, and at this stage, we assume the `WrapCnt` and `WrapSize` are big enough to accommodate all the trace data.

The second argument is a callback entry-point, from where the profiling starts. The third argument is a list of module names whose functions (both exported functions and local functions) will be traced. No functions will be traced if this list is empty.

To illustrate how the tool works, let's take a *similar code detection* algorithm for Erlang programs as an example. The similar code detection program takes a list of directories/files and some threshold values as parameters, and returns the similar code fragments found in those Erlang files.

We could use the following command to start the profiling of the similar code detection process:

```
percept2:profile("sim_code.dat",
                 {sim_code,sim_code_detection,
                 [["./test"], 3, 40, 2, 4, 0.8, [], 8]},
                 [sim_code]).
```

In this example, we choose to trace all the functions defined in the module `sim_code`, which implements the clone detection algorithm.

Percept2 sets up the trace and profiling facilities to listen for the traced events. It then stores these events to the file: `sim_code.dat`. Alternatively, we could use the following command to have multiple data files to store the trace data:

```
percept2:profile({"sim_code", wrap, ".dat", 20000000, 10},
                 {sim_code,sim_code_detection,
                 [["./test"], 3, 40, 2, 4, 0.8, [], 8]},
                 [sim_code]).
```

In the latter case, Percept2 stores trace events to files: `sim_code0.dat, sim_code1.dat`, etc. The actual run of the profiling command would generate 4 data files: i.e. `sim_code0.dat, sim_code1.dat, sim_code2.dat` and `sim_code3.dat`.

The profiling will go on for the whole duration until the function `sim_code:sim_code_detection/8` returns and the profiling has concluded.

4.2.2 Analysing

To analyze the data files generated, we use the function `percept2:analyze/3`, which takes a list of data file names as input. This function parses the data files in parallel, and inserts all events into a RAM database. To analyse the trace data from the previous example, we run the following command:

```
(test@hl-lt)2> percept2:analyze(["sim_code0.dat", "sim_code1.dat",
                                 "sim_code2.dat", "sim_code3.dat"]).
Parsing: "sim_code0.dat"
Parsing: "sim_code1.dat"
Parsing: "sim_code2.dat"
Parsing: "sim_code3.dat"
```

```
Parsed 83686 entries from "sim_code0.dat" in 7.191 s.
Parsed 129217 entries from "sim_code1.dat" in 9.064 s.
Parsed 25927 entries from "sim_code3.dat" in 10.78 s.
Parsed 128830 entries from "sim_code2.dat" in 10.796001 s.
Consolidating...
356 created processes.
97 opened ports.
ok
```

4.2.3 Data viewing

To view the data, start the web-server with `percept2:start_webserver/1` with a specific port number as the argument as shown next, or with `percept2:start_webserver/0` in which case an available port number will be assigned by inets. This command returns the hostname and a port where we should direct the web browser.

```
(test@hl-lt)3> percept2:start_webserver(8888).
{started,"hl-lt",8888}
```

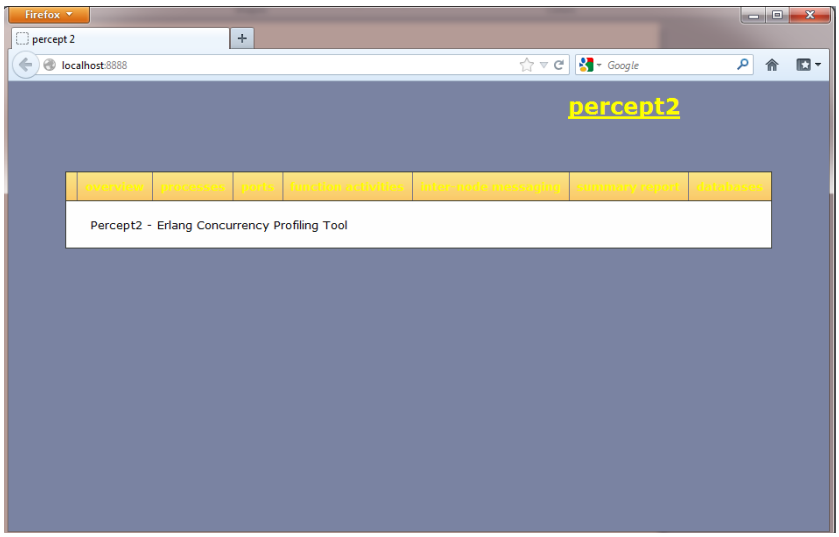

Figure 6: Percept2: menu page

As with Percept, we can now open a web-browser, and go to localhost with the port number returned; then we should be able to see the menu page of Percept2, as shown in Fig 6.

Same as Percept, clicking on the `overview` button in the menu will direct to the concurrency graph page as shown in Fig 7. Again, we can zoom in on different areas of the graph either using the mouse to select an area or by specifying min and max ranges in the edit boxes.

Selecting `schedulers` from the droplist option next to the `update` button will lead to the scheduler activity graph page. The scheduler activity graph shows the number of active schedulers at any time during the profiling, as shown in Fig 8.

As in Percept, clicking on the `processes` button in the menu bar will direct us to the processes view. Different from Percept, Percept2 shows the processes in an expandable/collapsible tree structure, as shown in Fig 9. A number of new process information items were added in Percept2:

- the column `#RQ_chgs` shows the number of times a process migrated between run-queues during the profiling. The actual migration history from one run queue to another is available from the process info information, which can be directed to by clicking on the process id;

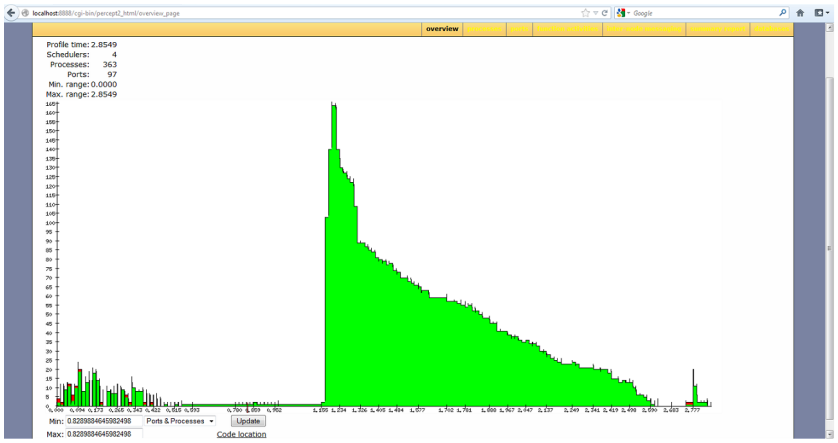

Figure 7: Percept2: concurrency overview

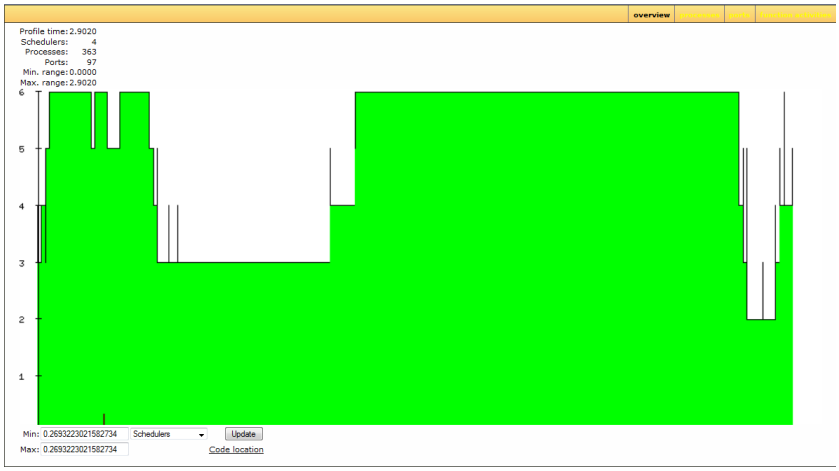

Figure 8: Percept2: scheduler graph

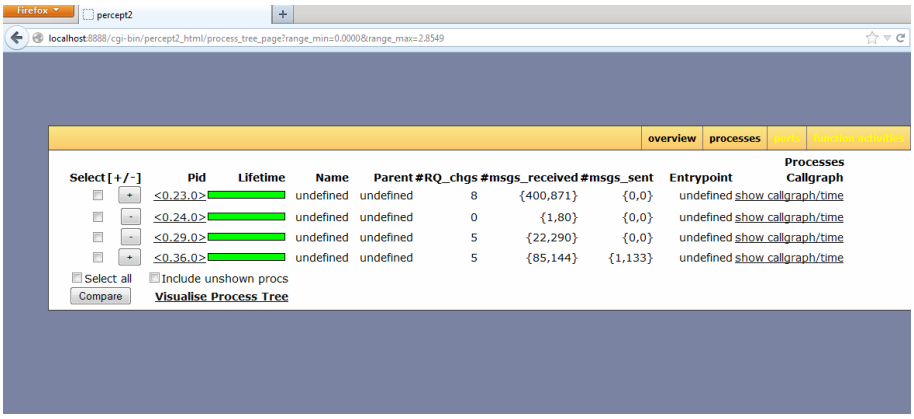

Figure 9: Percept2: process page (1)

- the column `#msgs_received` shows the number of messages received by a process (the first element of the tuple), as well as the average size of all these messages received (the second element of the tuple);
- similarly, the column `#msgs_sent` shows the number of messages sent by a process, as well as the average size of all these messages sent.

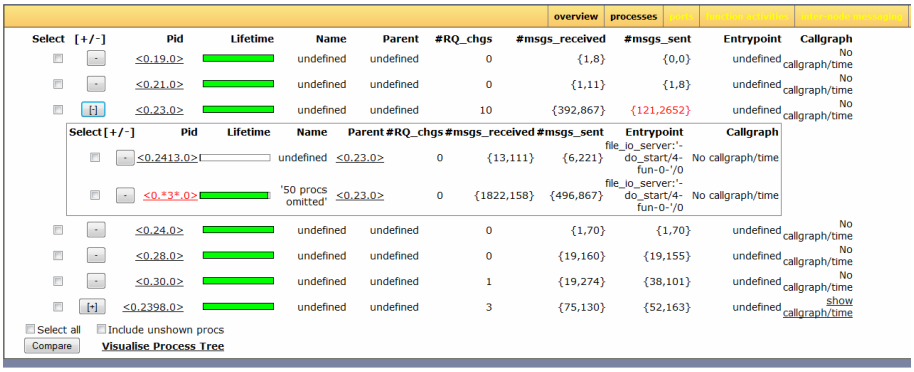

Figure 10: Percept2: process page (2)

Fig 10 shows the process tree when the + sign next to the Pid `<0.23.0>` is clicked. This snapshot shows that 51 processes were spawned by process `<0.23.0>`. In this case, all the 51 processes have the same entry point. Instead of listing all these 51 processes, Percept2 only shows information about one process, `<0.2413.0>` in this case, and all the remaining process are compressed into one single dummy processes. Only unregistered processes with the same parent and the same entry function can be compressed, and if this happens, the values of '`#RQ_chgs`', '`#msgs_received`' and '`#msgs_sent`' are the sum of all the processes represented by the dummy process.

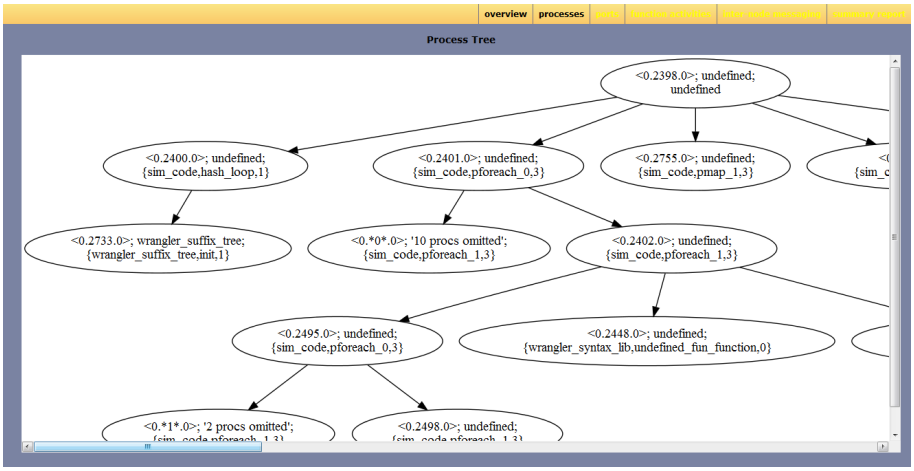

Figure 11: Percept2: process tree

If we click on the '*Visualise Process Tree*' link at the bottom of the table, a graph representation of all the process trees listed in the table will be shown as in Fig 11. Currently, the linkage relationships between parent/children processes are not reflected, but will be added in the future. Note that Percept2 use the *dot* command from *Graphviz* to generate the graph visualisation, so make sure that Graphviz works on your machine.

If functions are traced for a process, and those functions form a callgraph, then a '*show callgraph/-time*' link is shown in the Callgraph column of the process table; otherwise '*no callgraph/time*' is shown.

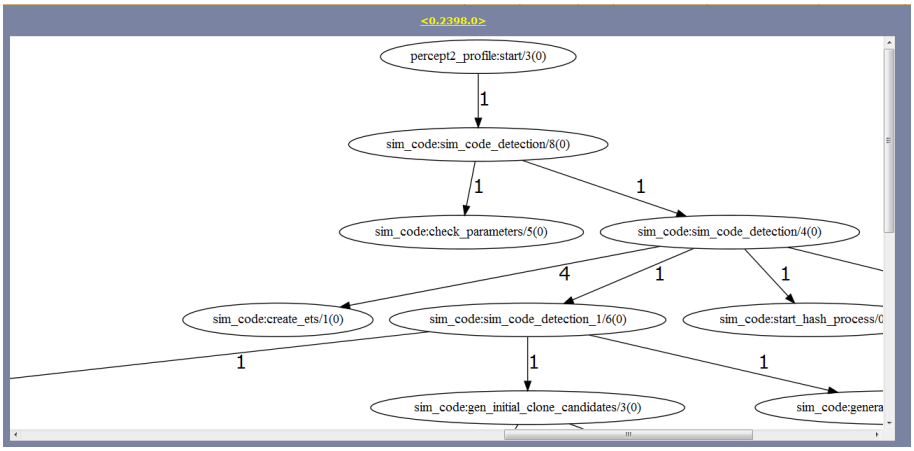

Figure 12: Percept2: function callgraph

Fig 12 shows the callgraph/time page of the process `<0.2398.0>`. In the callgraph shown, the edge label indicates how many times a function is called by its calling function during the profiling. Note that only functions that are actually traced are included in the callgraph, i.e. a function that is not traced is not included in the callgraph even if this function is called during the execution of the application.

Accumulated Calltime

module:function/arity	callcount	accumulated time
sim_code:sim_code_detection/8	1	2.856 98%
sim_code:sim_code_detection/4	1	2.839 98%
sim_code:sim_code_detection_1/6	1	2.839 98%
sim_code:generalise_and_hash_ast/6	1	2.667 92%
sim_code:gen_initial_clone_candidates/3	1	0.125 4%
sim_code:get_clone_candidates/3	1	0.109 4%
sim_code:check_clone_candidates/4	1	0.047 2%
sim_code:examine_clone_candidates/5	1	0.047 2%
sim_code:pforeach/2	2	0.047 2%
sim_code:get_final_clone_classes/1	1	0.000 0%
sim_code:process_initial_clones/1	1	0.000 0%
sim_code:combine_clones_by_au/1	1	0.000 0%
lists:map/2	1	0.000 0%
sim_code:combine_clones_by_au_1/1	1	0.000 0%
sim_code:pmap/2	1	0.000 0%
sim_code:'-pmap/2-lc$^0/1-0-'/1	1	0.000 0%
sim_code:'-pmap/2-lc$^1/1-1-'/3	1	0.000 0%
sim_code:stop_hash_process/1	1	0.000 0%
sim_code:stop_clone_check_process/1	1	0.000 0%
sim_code:start_hash_process/0	1	0.000 0%
sim_code:start_clone_check_process/1	1	0.000 0%
sim_code:'-check_clone_candidates/4-lc$^0/1-0-'/1	1	0.000 0%
sim_code:create_ets/1	4	0.000 0%
sim_code:check_parameters/5	1	0.000 0%
sim_code:'-process_initial_clones/1-fun-0-'/1	16	0.000 0%
percept2_profile:start/3	1	0.000 0%

Figure 13: Percept2: function calltime

Underneath the callgraph is a table, as shown in Fig 13, indicating the accumulated time in each function traced.

Function names shown in the accumulated calltime table are click-able, and clicking on a function name will direct us to the information page for this function. Fig 14 shows the information about the function `sim_code:sim_code_detection/4` executed by process `<0.2398.0>`.

Slightly different to Percept, Percept2 separates the display of ports information from that of processes. To get a more detailed description about ports, select the ports view by clicking on the ports

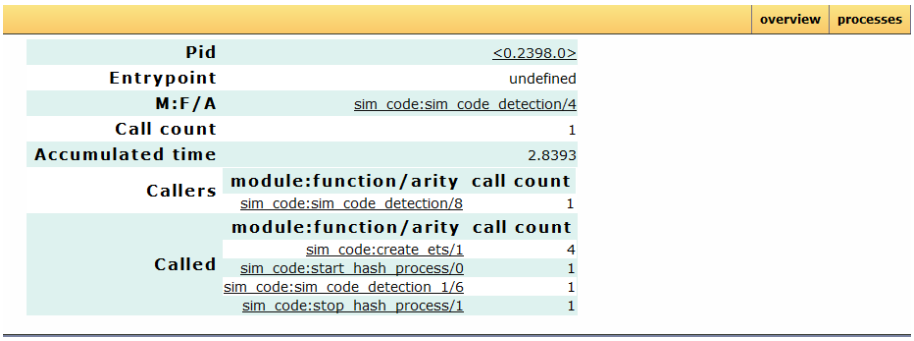

Figure 14: Percept2: function information page

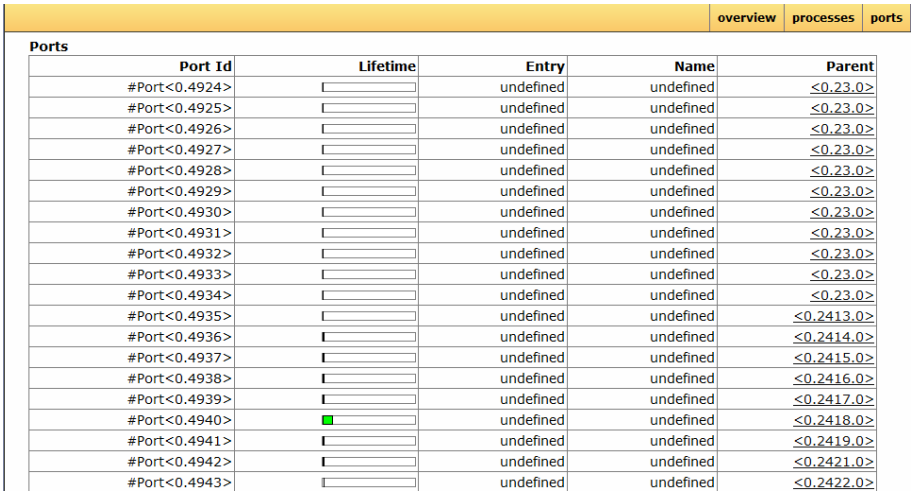

Figure 15: Percept2: ports information page

button in the menu. Information about the lifetime, parent process pid, etc, are shown in the table, as shown in Fig 15.

If functions are traced, clicking on the '*function activities*' button in the main menu will direct to a page showing the functions that are active during the time interval selected. A time interval is selected from the overview page either by selecting an area along the time line, or by specifying min and max ranges in the edit boxes, and then pressing the update button. The default time interval is the whole profiling period.

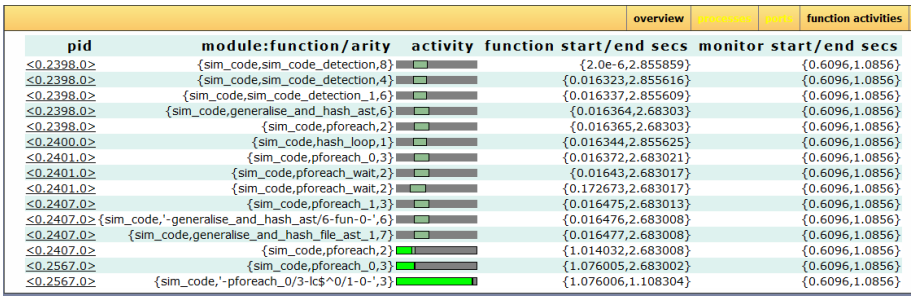

Figure 16: Percept2: function activities

As shown in Fig 16, the function activities table shows how the lifetime of an active function overlaps with the time interval selected. In the activity bar, the green part shows the time interval selected, light green shows the overlapping between the function's lifetime and the time interval selected. The grey part means that the function is active, but the time is out of the time interval selected.

Percept2 provides limited support for tracing distributed nodes so far, but one thing Percept2 can report is the message passing activities between nodes. The tracing of inter-node communication can be set up using Erlang's ttb/inviso library. Once the trace data has been collected, the trace files can be passed to Percept2, which will then analyze the data and extract those message passing activities between nodes.

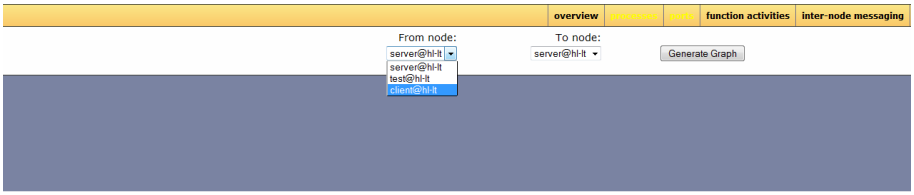

Figure 17: Percept2: inter-node messaging interface

When multiple nodes have been profiled, clicking on the '*inter-node messaging*' button in the menu will direct us to a page like the snapshot shown in Fig 17. From this page, we can select the two nodes that we are interested, `Node1` and `Node2` say, and click on the *Generate Graph* button, then Percept2 will generate a graph showing the message passing activities from `Node1` to `Node2` as shown in Fig 18. The X-axis of the graph represents the time line, and the Y-axis of the graph represents the size of the message sent.

5 Using DTrace/SystemTap

5.1 Technical issues

As DTrace/SystemTap were considered to be important for RELEASE from the early stages of the project, an effort was made early on to overcome technical difficulties and provide practical guidelines

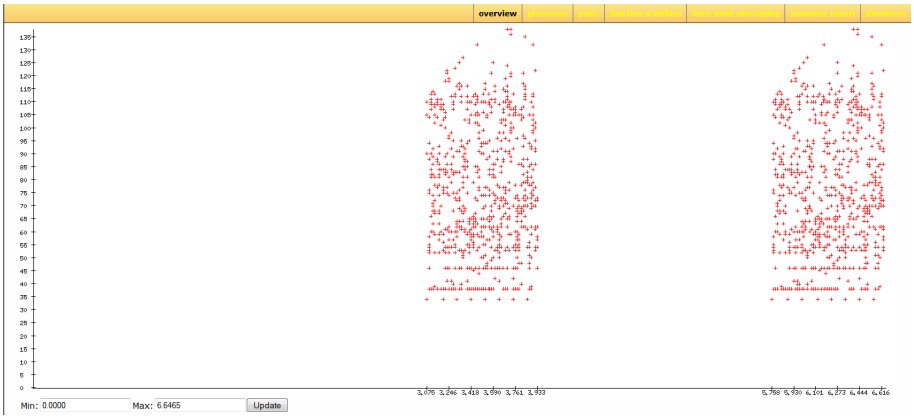

Figure 18: Percept2: inter-node messaging

that would allow their use by project members as easily as possible. DTrace runs primarily on Oracle Solaris, Mac OS X and FreeBSD. However, Solaris and FreeBSD are not very popular operating systems today and, although Mac OS X has a quite fat share in the market of desktops and laptops manufactured by Apple, it is not applicable to server machines with large numbers of cores that are of interest in the context of RELEASE. Therefore, we early turned our attention to Linux and SystemTap.

For the proper use of SystemTap on a system running Linux, two things are necessary: (i) the kernel needs to have debug information, and (ii) the kernel needs to have tracing support enabled (CONFIG_UTRACE). With the exception of very few Linux distributions (most notably, Fedora Core), the latter can be achieved by a special kernel patch (UTRACE). We have made available patched kernels (2.6.32) for the amd64 architecture and for GNU Debian Linux; such kernels have been installed and tested on various sites of RELEASE, mainly in Athens and Uppsala. The UTRACE kernel patch will no more be necessary when the Linux kernels used by mainstream distributions advance to 3.5.x, where UTRACE has been replaced by Uprobes, an alternative which is considered cleaner and is supposed to collaborate smoothly with DTrace/SystemTap.

5.2 Experimental tools

In order to get familiar with the DTrace/SystemTap profiling infrastructure, we initially tried to collect information about the sizes of the run queues of the available schedulers during the execution of an Erlang program, and to measure how often process migration takes place. To do this, we had to insert new probes in the Erlang VM. We then wrote DTrace and a SystemTap scripts (there are some minor syntactic differences between the languages used by these two tools, which are of little interest for this report) to inspect changes in the run queue sizes during a program's lifetime.

For this work, we focused on the following events: creation of a run queue, process enqueue/dequeue, process migration; thus, we track changes to run queues when the relevant probes fire (e.g., in case of SystemTap, those probes are: `run_queue__create`, `run_queue__enqueue`, `run_queue__dequeue`, `process__migrate`) and log them to a file at a constant interval (e.g., every second). Then, we process the log file with a few `awk` scripts and use `gnuplot` to visualize the results. Two examples of the figures that we are able to generate with these scripts are shown in Figures 19 and 20.

This was just done as an experiment of what kind of information we can get by using some of the available VM probes, by introducing new probes, and with a simple visualizer using the `gnuplot` utility. Then, our work focused on integrating DTrace/SystemTap as a back-end for collecting profiling data in `percept2`.

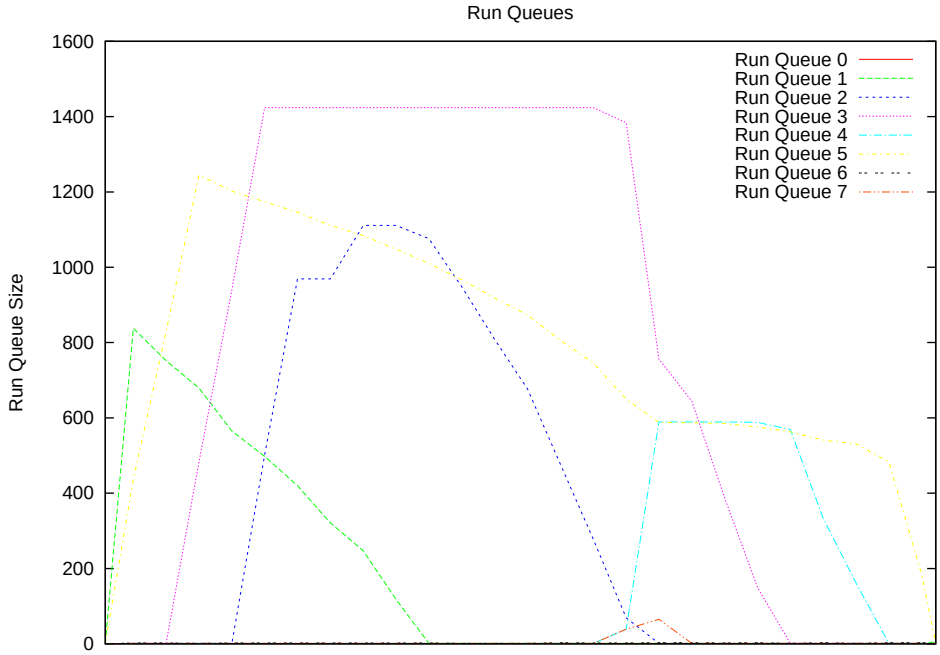

Figure 19: A simple graph of the changes in run queue size running `bang` with 8 schedulers

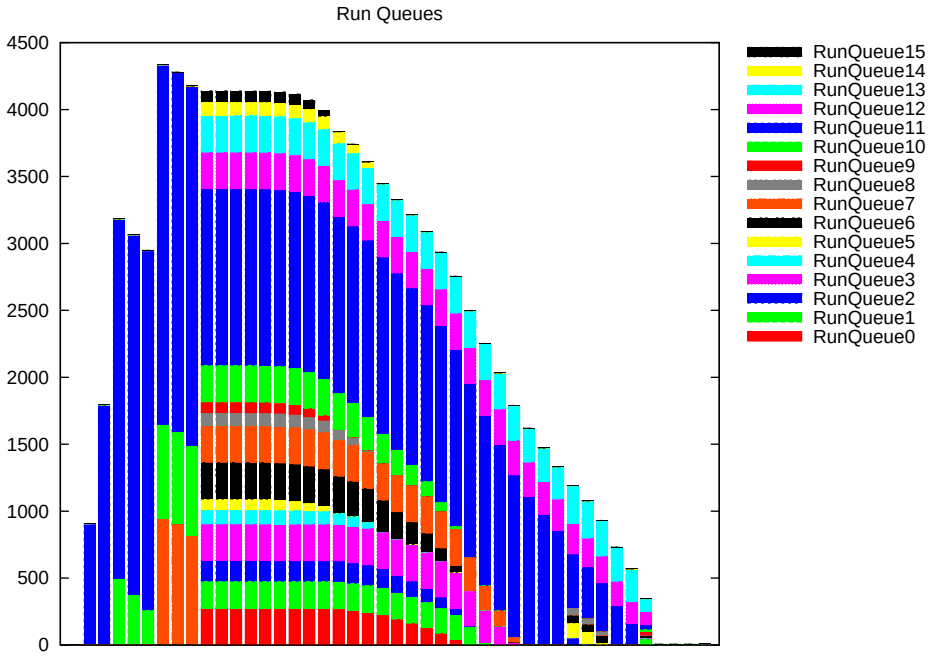

Figure 20: A stacked variable-height bar graph of the size of the run queues running `bang` with 16 schedulers

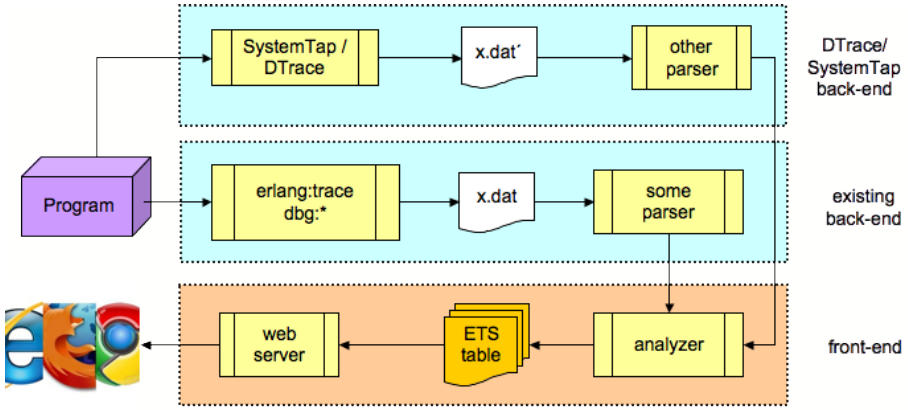

Figure 21: DTrace/SystemTap Percept

5.3 DTrace/SystemTap Percept

In this section we present the work we have done so far for designing and implementing a new back-end for *Percept* that uses DTrace or SystemTap in order to collect trace information, instead of the Erlang built-in tracing functions that the existing back-end of Percept uses.

Our main goal, when designing the DTrace/SystemTap back-end, was to try to re-use as much as possible from the existing infrastructure of Percept, both for saving effort and for compatibility reasons. In comparison with the existing back-end of Percept, the new back-end uses:

- a *different* mechanism for collecting information about Erlang programs,
- a *different* format for the trace file it produces,
- a *different* parser for parsing the trace file,
- the *same* storage infrastructure, and
- the *same* presentation facilities.

An overview of Percept after the addition of the new back-end is shown in Figure 21.

5.3.1 DTrace/SystemTap Percept by example

This section presents an example of how to use both the existing and the new back-end of Percept, in order to run an Erlang program and collect tracing information about it.

The scenario we are going to execute in both cases is the following:

- Start profiling
- Execute some piece of code
- Stop profiling
- Write all trace information that has been collected in a file
- View trace information

In order to implement this scenario using the existing back-end of Percept, we need to execute the following commands:

```
1> percept:profile("percept.dat").
Starting profiling.
{ok,#Port<0.657>}
2> bang:bang(10,10).
ok
3> bang:bang(20,20).
ok
4> percept_profile:stop().
ok
```

To analyze the collected information, we use the command:

```
1> percept:analyze("percept.dat").
Parsing: "percept.dat"
Parsed 876 entries in 0.019469 s.
Consolidating...
37 created processes.
0 opened ports.
ok
```

Finally, we use the command:

```
1> percept:start_webserver().
{started,"greedy",58141}
```

to start a web server. To view the data, we just need to direct a web browser to the hostname and port that the above command returned.

In order to execute the same scenario using the new DTrace/SystemTap back-end of Percept, we first need to execute the DTrace/SystemTap script that is responsible for collecting the trace information (in the form of fired DTrace probes):

```
dtrace dtrace-percept.d > dtrace_percept.dat
```

or

```
stap dtrace-percept.stap > dtrace_percept.dat
```

and redirect its output in the file, where we want all trace information to be written.

For the purposes of the new back-end, we specified a new DTrace probe that is fired at the exact same locations in the Erlang/OTP code, where trace information that Percept uses is collected.

```
probe percept__trace(char *)
```

The only argument of the new probe is a piece of trace information in the external term format.

After we have started the appropriate script, we can run the code we want to profile:

```
erl -noshell - eval "bang:bang(10,10). bang:bang(20,20)." -s init stop
```

Once the execution of the code has finished, we stop the DTrace or SystemTap script. All the necessary trace information should now be available in the trace file, where we redirected the script's output.

In order to parse and analyze this trace file, we need to execute the following command:

```
1> percept:d_analyze("dtrace_percept.dat").
```

`percept:d_analyze/1` is a function that is similar to the existing `percept:analyze/1` function: it parses the given trace file and inserts all trace information into an ETS table. The trace file is essentially a number of Erlang terms in the external term format separated with dots. Thus, the parsing phase uses the `file:consult/1` and the `erlang:binary_to_term/1` functions.

In order to inspect the collected data, we use the `percept:start_webserver/0` function, as we did before.

6 Other Contributions

6.1 Sampling-based profiling

Complement to Percept2, we have also implemented a collection of functions for reporting information regarding memory usage, garbage collection, scheduler utilization, and message/run queue length, etc. This is done by sampling-based profiling, i.e. the profiler probes the running Erlang system at regular intervals. Sampling profiling is typically less numerically accurate and specific, but has less impact on the system. Data collected by the profiler are stored in files, and the `gnuplot` tool can be used for graph visualisation of the data. The following Erlang functions are used for the purpose of data collection `erlang:statistics/1`, `erlang:memory/1`, `erlang:system_info/1` and `erlang:process_info/1`. The interface functions for invoking the sampling-based profiling are `percept2:sample/3,4,5` and `percept2:sample/4` and `percept2:sample/5`.

The following types of information can be collected:

- Total run queue length: the sum length of all run queues, that is, the total number of processes that are ready to run.
- Length per run queue: the length of each run queue, that is, the number of processes that are ready to run in each run queue.
- Scheduler utilisation: the scheduler-utilisation rate per scheduler.
- Schedulers online: the amount of schedulers online.
- Process count: the number of processes currently existing at the local node.
- Memory information: information about memory dynamically allocated by the Erlang emulator. Information about the following memory types is collected: processes, ets, atom, code and binary.
- The number of messages currently in the message queue of a particular process.

Profiling data is formatted in the way so that the graph plotting tool Gnuplot can be used for visualisation. A pre-defined plotting script is available for each type of information collected, and these scripts are in the `gplt` directory user `percept2`. For more information about how to run the profiling and how to visualise the profiling data using Gnuplot, please refer to the Percept2 documentation.

6.2 Experimental extensions to the Erlang built-in trace

One of the factors that limit the scalability of tracing/profiling tools is the vast amount of data generated/collected, hence reducing the data generated/collected could potentially improved the scalability of tracing/profiling tool. For this purpose, we have carried out a couple of experimental extensions to the Erlang built-in trace. More details follow.

6.2.1 Message size vs. message

The current Erlang built-in tracing logs the complete messages *sent/received* by traced processes when the *send/receive* flags are on. While message content may be useful for some cases, it could also increase the size of trace data collected significantly, especially when large messages are sent between processes, or there are frequent message passing between processes.

We have extended the built-in trace with a flag to be used in conjunction with the *send/receive* trace flags, so that the size of the message, instead of the actual message, is logged when this flag is on.

6.2.2 Dynamic trace data filtering

Erlang's built-in tracing provides powerful support for controlling the function call events to traced by means of *match specification*. A match specification consists of an Erlang term describing a small program that expresses a condition to be matched over a set of arguments. When tracing is concerned, match specifications are used to deal with the filtering and manipulation of trace events. If they match successfully, a trace event is generated and some predefined actions can be executed. Match specifications are compiled to a format close to the one used by emulator, making them more efficient than functions.

While match specifications can be used to filter out function call related trace events, they are not applicable to other trace events, but this does not mean that there is no such a need for dynamic filtering/manipulating other trace event, actually quite the contrary.

To support dynamic filtering of non-function related trace events, we proposed to allow users to use match specifications to match general trace messages, and filter out those events not of interest. For this purpose, we have experimentally implemented a new Erlang BIF: `erlang:trace_filter/2`, which has the following type specification:

```
erlang:trace_filter(PidSpec, MatchSpec)->integer()>=0.

Types:
    PidSpec = pid() |existing | new | all
    MatchSpec = true | false | [match_specification()]
    match_spec(): see the ERTS User's Guide for a description of
                  match specifications.
```

`erlang:trace_filter/2` works like this: for all the trace events for the process or processes represented by `PidSpec`, the trace message is matched over the match specifications specified by `MatchSpec`, and a trace event is logged only if the match succeeds. Setting `MatchSpec` to `true` does not filter out any trace messages, and setting it to `false` will filter out all the trace messages, i.e. no trace messages for the processes specified will be recorded.

7 Future plans

In the second year of the project we will be developing tools for support of online monitoring and visualisation of SD Erlang programs, while at the same time enhancing the offline monitoring tools presented here. In particular we plan to explore

- Enhancing Percept2 so that it acts as a single front end for the results of both Erlang trace and DTrace/SystemTap.
- To include further support for distribution in Percept2.
- Explicitly to support the constructs of SD Erlang in offline and online visualisation.

- To work with experts in the field of data visualisation from the University of Kent's Computational Intelligence research group to develop innovative approaches to HTML5 for web-based graphical visualisation.

8 Conclusions

This report describes the first deliverable in the fifth work package of the RELEASE project, and delivers a number of tools and mechanisms supporting the offline visualisation of SD Erlang many/multicore systems.

References

[Arm10] J. Armstrong. Erlang. *Commun. ACM*, 53:68–75, 2010.

[Bul08] G. Bulmer. Erlang-DTrace. Presented at the Erlang User Conference, November 2008.

[cpr] cprof - A simple Call Count Profiling Tool. `http://www.erlang.org/doc/man/cprof.html`.

[CT09] F. Cesarini and S. Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media Inc., 1st edition, 2009.

[dbg] dbg - The Text Based Trace Facility. `http://www.erlang.org/doc/man/dbg.html`.

[epr] eprof - A Time Profiling Tool for Erlang. `http://www.erlang.org/doc/man/eprof.html`.

[et] ET - The Event Tracer. `http://www.erlang.org/doc/man/et.html`.

[eto] ETop - The Erlang Top. `http://www.erlang.org/doc/man/etop.html`.

[fpr] fprof - An Erlang File Trace Profiler. `http://www.erlang.org/doc/man/fprof.html`.

[Fri11] S. L. Fritchie. DTrace and Erlang: A new beginning. Presented at the Erlang User Conference, November 2011.

[GM11] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[lcn] lcnt - A runtime system Lock Profiling tool. `http://www.erlang.org/doc/man/lcnt.html`.

[per] Percept - An Erlang Concurrency Profiling Tool. `http://www.erlang.org/doc/man/percept.html`.

[pma] pman - A Graphic Process Manager. `http://www.erlang.org/doc/man/pman.html`.

[ttb] TTB - The Trace Tool Builder. `http://www.erlang.org/doc/man/ttb.html`.